## Vector architectures

Vector architectures grab sets of data elements scattered about memory, place them into large, sequential file registers, operate on data in those register files and then disperse the results back into memory: a single instruction operates on *vector data*, which results in dozens of register-register operations on independent data elements.

These large register files act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth. Since vector loads and stores are deeply pipelined, the program pays the long memory latency only once per vector load or store versus once per element, thus amortising the latency over, for example, 64 items.

### The VMIPS architecture

In order to see how vector architectures are laid out, we are now going to take a look at a vector architecture based on the standard MIPS pipeline: we will call this ISA *VMIPS*.

The primary components of the VMIPS architecture are:

- **Vector registers**: each vector register is a fixed-length band holding a single vector. VMIPS has eight vector registers, each of which holds 64, 64-bit wide elements.
- **Vector functional units**: each functional unit is fully pipelined and it can start a new operation on every clock cycle. A control unit is needed to detect hazards, both for structural ones on FUs and data ones on register accesses.
- **Vector load-store unit**: the vector memory unit loads or stores a vector to or from memory. The VMIPS vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency. This unit would also normally handle scalar loads and stores.
- **A set of scalar registers**: scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load-store unit. These are the normal, 32 general-purpose registers and 32 floating-point registers of MIPS.

Vector architectures like VMIPS have specific instructions for vector operations. In the VMIPS case, vector operations use the same names as scalar MIPS instructions,

but with the letters "VV" appended. Thus, `ADDVV.D` is an addition of two double-precision vectors.

Vector instructions take as their input either a pair of vector registers, as we've seen with `ADDVV.D`, or a vector register and a scalar register: the instruction `ADDVS.D` takes a vector and adds the scalar value passed to it to every vector element.

The names `LV` and `SV` denote vector loads and stores and move double precision vectors from and to memory respectively.

### How vector processors work: an example

We can best understand a vector processor by looking at a vector loop for VMIPS. Let's assume we want to perform the following computation:

$$Y = a \times X + Y$$

$X$ and $Y$ are vectors, initially resident in memory, and $a$ is a scalar. This is the so-called *SAXPY* or *DAXPY* loop that forms the inner loop of the Linpack benchmark. SAXPY stands for single-precision $a \times X + Y$, while DAXPY is the double-precision variant.

Let's assume that the number of elements of a vector register matches the length of the vector operation we are interested in for this example.

In a standard MIPS architecture, the code to execute DAXPY would be as follows:

```
        L.D    F0,  a        ;load scalar a
        DADDIU R4,  Rx, #512 ;last address to load
Loop:   L.D    F2, 0(Rx)     ;load X[i]
        MUL.D  F2,  F2, F0   ;a * X[i]
        L.D    F4,  0(Ry)    ;load Y[i]
        ADD.D  F4,  F4, F2   ;a * X[i] + Y[i]
        S.D    F4,  9(Ry)    ;store into Y[i]
        DADDIU Rx,  Rx, #8   ;increment index to X
        DADDIU Ry,  Ry, #8   ;increment index to Y
        DSUBU  R20, R4, Rx   ;compute bound
        BNEZ   R20, Loop     ;check if done
```

While in VMIPS, the code is far simpler:

```
L.D     F0, a      ;load scalar a
LV      V1, Rx     ;load vector X
MULVS.D V2, V1, F0 ;vector-scalar multiplication
LV      V3, Ry     ;load vector Y
ADDVV.D V4, V2, V3 ;vector-vector addition
SV      V4, Ry     ;store result
```

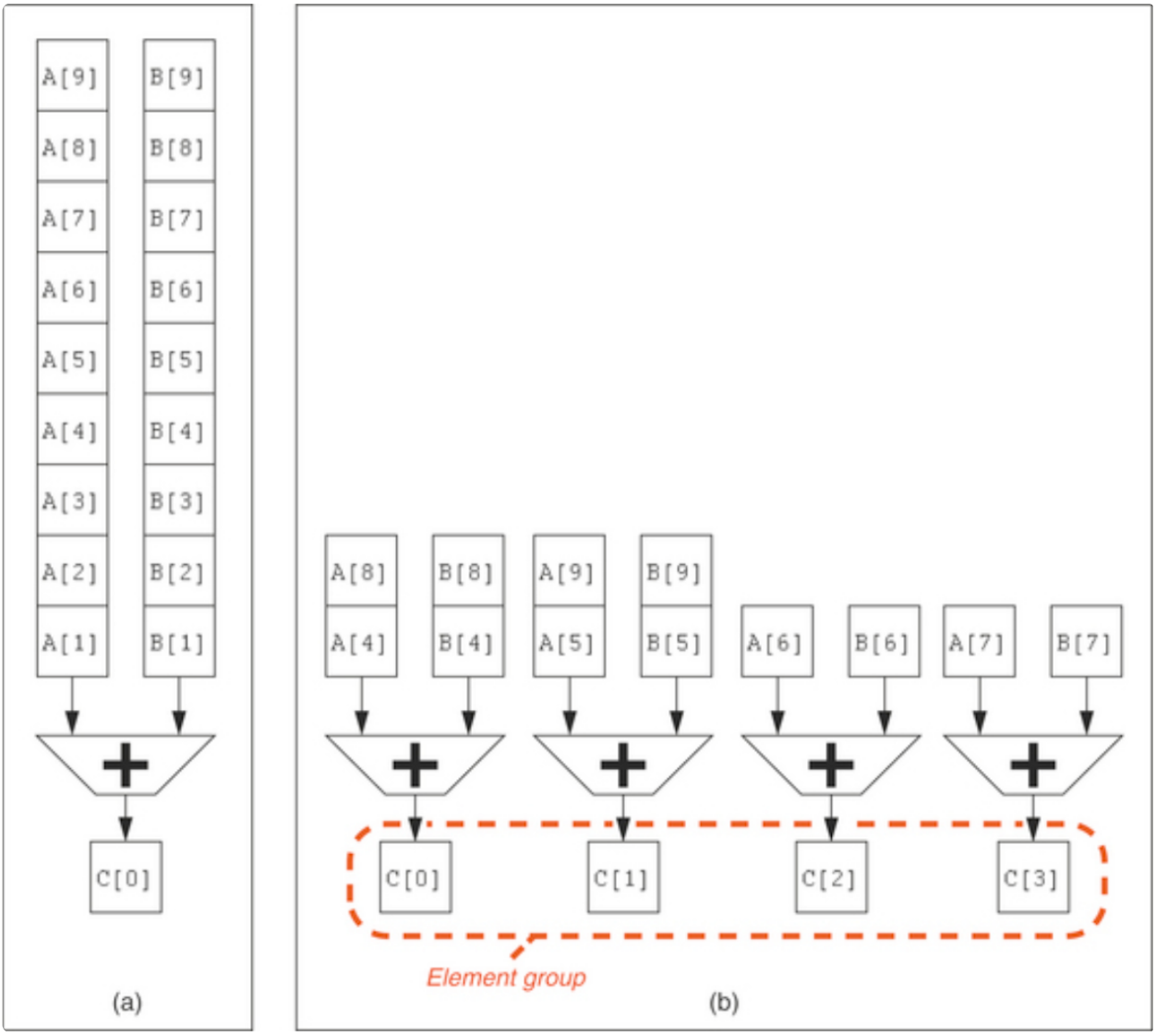
The loop is gone and the number of instructions is reduced from almost 600 in the MIPS variant to just 6 in the VMIPS program. This is thanks to the fact that each individual element of the vectors is independent from the others, meaning that *loop-carried dependencies*—the dependencies between two iterations of a loop—are absent, and thus all elements can be worked on independently. This is the type of application in which vector architectures shine.

### Multiple lanes: beyond one element per clock cycle

A critical advantage of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single, short instruction. A single vector instruction can include scores of independent operations yet be encoded in the same number of bits as a conventional scalar instruction.

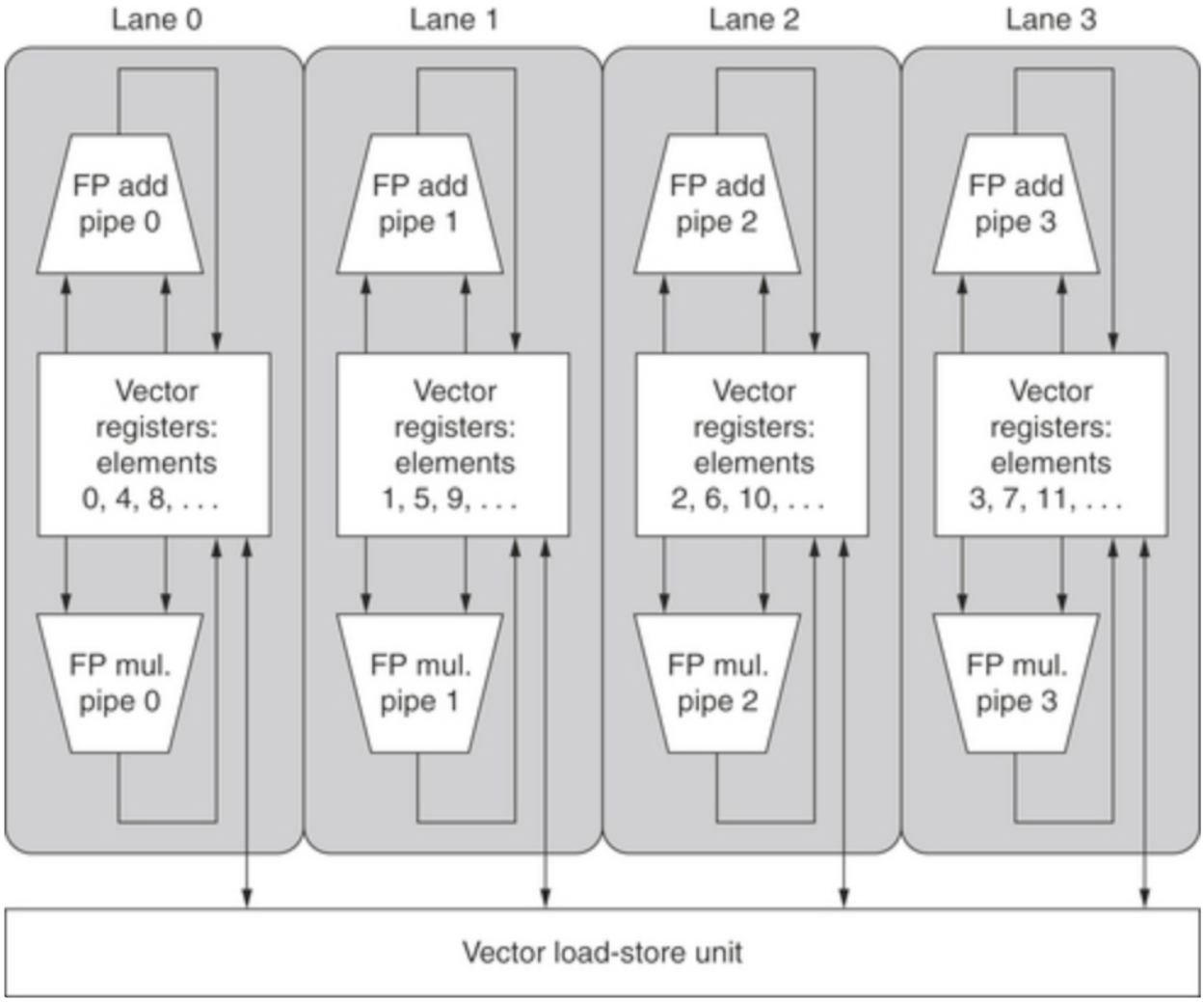
The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using a deeply pipelined functional unit, as in the VMIPS implementation we've seen so far, an array of parallel functional units or a combination of parallel and pipelined functional units.

The following image illustrates how an operation such as $C = A + B$, where $A, B, C$ are all vectors, can be executed very quickly in a vector architecture:




The VMIPS instruction set has the property that all vector arithmetic instruction only allow element $N$ of one vector register to take part in operations with element $N$ from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as **multiple lanes**.

We can thus increase the peak throughput of our vector processor by adding more lanes:




Each lane contains one portion of the vector register file and one execution pipeline for each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane. The first lane holds the first element for all vector registers, and so destination operands located in the first lane. This allocation allows the arithmetic pipeline local to the lane to complete the operations **without communicating with other lanes**. Avoiding interlane communication reduces the wiring cost and register file ports required to build a highly parallel execution unit and helps explain why vector computers can complete up to 64 operations per clock cycle.